Expressing Advanced User Preferences in Component Installation[*]

Ralf Treinen
Université Paris Diderot
PPS, UMR 7126, France
Ralf.Treinen@pps.jussieu.fr

Stefano Zacchiroli
Université Paris Diderot
PPS, UMR 7126, France
zack@pps.jussieu.fr

ABSTRACT

State of the art component-based *software collections*—such as FOSS distributions—are made of up to dozens of thousands components, with complex inter-dependencies and conflicts. Given a particular installation of such a system, each *request* to alter the set of installed components has potentially (too) many satisfying answers.

We present an architecture that allows to express advanced user preferences about package selection in FOSS distributions. The architecture is composed by a distribution-independent format for describing available and installed packages called CUDF (Common Upgradeability Description Format), and a foundational language called MooML to specify optimization criteria. We present the syntax and semantics of CUDF and MooML, and discuss the partial evaluation mechanism of MooML which allows to gain efficiency in package dependency solvers.

Categories and Subject Descriptors

K.6.3 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Software Management—*Software selection*; D.2.9 [**SOFTWARE ENGINEERING**]: Management—*Life cycle*

General Terms

Design, Languages, Management

Keywords

FOSS, upgrade, packages, selection, preferences

1. INTRODUCTION

One of the noteworthy characteristics of FOSS (for Free and Open Source Software) distributions—such as Debian GNU/Linux, Red Hat Enterprise Linux, or FreeBSD—is the availability of large numbers of components (usually called *packages* in this environment) that can be installed, removed, and upgraded as single entities. Systems like Debian can have up to dozens of thousands components, growing steadily across releases and linked by complex inter-dependencies [1]. Similar architectures exist in other contexts where components are used to define the granularity at which software can be deployed: the analogous of FOSS packages can be found for example in the Eclipse [3] and Maven[1] platforms; in both cases the number of components and their inter-relationships are similar to what exists in common FOSS distributions.

In all such scenarios, user installations are managed using tools such as *package managers* which receive user requests to change the installation in some way—e.g. install a new component—and try to satisfy them equipped with the knowledge of where to find components and which are their inter-relationships. When the number of components grows, a given user request can have thousands of acceptable solutions. For instance, in satisfying the simple "install wordpress" request a package manager can be faced with questions like: "which version of wordpress should be installed?", "using which web server?", "relying on which PHP implementation", etc. The number of potential solutions for the final user can easily grow exponentially; currently, the actual choice depends on internal heuristics implemented by specific package managers and is customizable in ad-hoc ways.

This paper focuses on FOSS distributions and presents an architecture to specify advanced user preferences in that context, abstracting over package manager specific details. The architecture is composed by two parts: a format to describe upgrade scenarios called CUDF (Common Upgradeability Description Format) and a foundational language to encode user preferences called MooML (MancOosi Optimization Meta-Language).

The MooML language is *foundational* in the sense that it is not (necessarily) meant to be a language for the end user or the system administrator; it is rather meant as an intermediate language with a precise semantics, which can be used by developers of installation tools as an abstract input language for expressing user preferences, and which on the other hand can be the target language for representing the choice a user may have expressed, for instance using some graphic interface. Mancoosi, which gives the name to MooML, is an ongoing project which aims, among oth-

[*]Partially supported by they European Community's 7th Framework Programme (FP7/2007–2013), Mancoosi project, grant agreement n. 214898.

[1]http://maven.apache.org/

ers, to develop better algorithms and tools to plan upgrade paths based on various information sources about software packages and on optimization criteria [6].

Paper structure.

The remainder of this section outlines the upgrade process FOSS packages are subject to. Section 2 presents common optimization criteria for package upgrade scenarios; there criteria will serve as running examples throughout the paper. Section 3 summarizes essential features of the CUDF language for describing upgrade scenarios. MooML itself and its partial evaluation mechanism are presented respectively in Section 4 and 5.

1.1 FOSS Package Upgrade Generalities

Packages.

FOSS (binary) distributions are organized as collections of *packages*, i.e. abstractions defining the granularity at which users can act (add, remove, upgrade, etc.) on installed software. Abstracting over format-specific details, a *package* is a bundle of the 3 parts depicted in Figure 1.

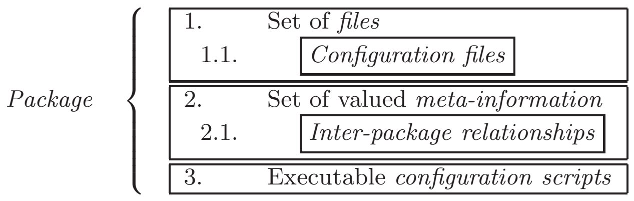
Package
1. Set of *files*
1.1. *Configuration files*
2. Set of valued *meta-information*
2.1. *Inter-package relationships*
3. Executable *configuration scripts*

Figure 1: Constituents of a package.

The set of files (1) represents what the package is delivering: executable binaries, data, documentation, etc. This set includes configuration files (1.1), that affect the runtime behavior of the package and are meant to be locally customized. Package meta-information (2) contains information varying from distribution to distribution. A common core provides: its name (a unique identifier), a version (taken from a totally ordered set), maintainer and package description, and most notably *inter-package relationships* (2.1). The kinds of relationship vary with the package manager used, but there exists a de facto common subset including dependencies (the need of other packages to work properly), conflicts (the inability of being co-installed with other packages), feature provisions (the ability to declare named features as provided by a given package, so that other packages can depend on them), and restricted boolean combinations of them [8]. Finally, packages come with a set of executable configuration (or *maintainer*) scripts (3). Their purpose is to let package maintainers attach actions to hooks executed by the installer; actions are used to finalize package configuration during deployment.

Upgrades.

A *distribution* is a collection of packages. The subset of a distribution corresponding to the packages actually installed on a machine is called *package status* and is meant to be altered using a *package manager*. An *upgrade scenario* is the situation in which a user, typically the system administrator, submits a *user request* to the package manager, with the intention to alter the packages status. Several entities and problems are involved in, and should be grasped by a complete description of, an upgrade scenario [7]. The main entities are packages and the most relevant problem for the present paper is *upgrade planning*; both are briefly described below.

```
 # apt-get install aterm
 Reading package lists... Done
 Building dependency tree... Done
 The following extra packages will be installed:
   libafterimage0
 0 upgraded, 2 newly installed, 0 to remove and
   1786 not upgraded.
 Need to get 386kB of archives.
 807kB of additional disk space will be used.
 Get: 1 http://ftp.debian.org libafterimage0 2.2.8
 Get: 2 http://ftp.debian.org aterm 1.0.1-4
 Fetched 386kB in 0s (410kB/s)
 Selecting package libafterimage0.
 (Reading database ... 294774 files and dirs ...)
 Unpacking libafterimage0 ...
 Selecting package aterm.
 Unpacking aterm (aterm_1.0.1-4_i386.deb) ...
 Setting up libafterimage0 (2.2.8-2) ...
 Setting up aterm (1.0.1-4) ...
```

Table 1: The package upgrade process. Horizontal lines separate the phases described in the text.

Table 1 summarizes the different phases of the *upgrade process*, using as an example the popular `apt-get` package manager (others follow a similar process). Phase (1) is a user specification of how the package status should be altered. The expressiveness of the request language varies with the package manager: it can be as simple as requesting the installation/removal of a single package by name, or can also enable limited expression of per-package preferences such as APT pinning [11]. Phase (2) (dependency resolution) checks whether a package status satisfying all dependencies and user request exists, it has been shown that this problem is at least NP-complete [8]. If this is the case, one such package status is chosen—trying to satisfy user preferences, if any—and gets called *solution*. Deploying a new status corresponding to the solution consists of package retrieval (3) and unpacking (4), possibly intertwined with several configuration phases (5) where maintainer scripts get executed.

Various challenges related to the upgrade process still need to be properly addressed. An example of a very practical challenge is the need to provide *transactional upgrades*, offering the possibility to roll back in case an unexpected (and unpredictable in general) failure is encountered during upgrade deployment [7]. Other challenges concern *upgrade planning*. For instance, dependency resolution can fail either because the user request is unsatisfiable (e.g., user error or inconsistent distributions [9]) or because the package manager is unable to find a solution. Completeness—the guarantee that a solution will be found whenever one exists—is a desirable package manager property [15], unfortunately missing in most package managers, with too few claimed exceptions [10, 17].

User Preferences.

While suitable and complete techniques to provide dependency solving completeness are now well-known [9] and "just" lack widespread adoption, *handling of complex user* *preferences* is a novel problem for software upgrade, and is the main concern of this paper. It boils down to let users specify what constitutes the "best" solution among all acceptable solutions, and provide mechanisms to efficiently find it. Example of preferences are *policies* [10, 16], like minimizing the download size or prioritizing popular packages, and also more specific requirements such as blacklisting packages maintained by an untrusted maintainer.

The first necessary step to attack the problem is devising a way to encode user preferences in a flexible way, without hindering package manager ability to respect them. A prerequisite of that is a rigorous description of upgrade scenarios, on top of which the *meaning* of user preferences will be defined.

2. USER PREFERENCE SCENARIOS

In the following we will consider several possible scenarios where user needs can be better encoded as user preferences in MooML. The actual encoding in MooML will be presented in Section 4.2, after a more in depth presentation of the language.

Size Minimizing the total size consumed by the package installation is a rather most basic optimization criterion and a frequent need of package managers for embedded systems.

Freshness Preferring more recent package versions over older package versions is also very common, and hard-wired in most package managers. The hard-wiring in Debian's APT as a *hard* constraint is the main cause for the incompleteness of its dependency solving abilities.

Pinning To avoid forcing the choice of the most recent version of a package in all cases, APT enables to specify different choices for specific packages by the mean of a mechanism called *pinning* [11]. In its essence, pinning consists in specifying integer score values (called *priorities*) for individual packages based on patterns of package names, package versions, and origin; among all the versions of a given package, the one with the highest priority gets chosen. By default, priority follows versions. This is an example of "local" preferences that apply to particular packages, in contrast to uniform constraint like total installation size.

Security updates usually should have highest priority while choosing which packages have to be upgraded. We will demonstrate MooML's multi-criteria capabilities by stating that maximizing security updates has priority over package freshness.

Multiple packages Some package managers, most notably `rpm`, allow for multiple versions of the same package to be installed; while this is an interesting property, one might want to automatically "clean up" useless multiple installations. This scenario will show how to minimize the number of packages that are installed in multiple versions.

Note that, while they are presented as such for the sake of brevity, scenarios are not mutually exclusive in practice. In our vision, some optimization criteria will constitute a default configuration of a given package manager (e.g.: always prioritizing security upgrades, avoiding package downgrades, etc.) while some other will be added by users by the mean of specific user interfaces. Even when the latter possibility is not exploited, there are advantages in externalizing preferences which are currently hard-wired in solving algorithms: for instance they will become overwritable by users and it will be easier to share optimizers among distributions.

MooML allows to combine multiple optimization criteria, however one has to specify a hierarchy among the multiple critera. For instance one can require to search for a solution that is minimal in size first, and among all that solutions that are minimal in size to choose one with maximal freshness. It is not possible to optimize two independent criteria at the same time since in that case an optimal solution might not exist.

As next section will explain, optimization criteria do not allow to taint the correctness of a solution, e.g. by allowing to install at the same time two conflicting packages.

3. DESCRIBING UPGRADE SCENARIOS

State of the art mechanisms for specifying user preferences highlighted so far [10, 17, 11] suffer from two main drawbacks: they are package manager specific, and they are not expressive enough to encode all our user preference scenarios (see Section 2). The first step we pursue in addressing these shortcoming is devising a rigorous format in which upgrade scenarios can be encoded; a user preference language will be then developed on top of such a format (see Section Section 4). The format is called CUDF (Common Upgradeability Description Format). The specification of CUDF [14] had been guided by some general *design principles*:

Be distribution agnostic One of the main purposes of CUDF is being a *common* format to encode upgrade scenarios coming from heterogeneous environments. As a consequence, CUDF is agnostic to distribution specific details such as the used package system or package manager.

Stay close to the original problem While there are several possible encoding of upgrade scenarios [9], CUDF aims to be as close as possible to the original problem, in order to preserve the ability for humans to understand the pre-CUDF upgrade scenario, and ease interoperability with legacy package managers.

Extensibility Core package properties—e.g.: name, version, dependencies, . . . —are shared by all distributions and essential to grasp the meaning of upgrade scenarios. Other auxiliary properties are not, but might be the subject of user preferences (e.g., minimize the number of "buggy" packages, according to distribution specific buggyness notions). In order not to hinder the possibility to express such user preferences on top of CUDF, the format allows to specify *extra package properties* not prescribed by the format specifications.

Transactional semantics The point of view of CUDF is upgrade planning: the notion of correctness of a solution with respect to an upgrade scenario expressed in CUDF is global and does not express the package deployment steps needed to pass from the starting package status to the final one. Such steps are more low-level, and mostly uninteresting for user preferences.

Plain text format Technically, CUDF aims at being simple to parse and to generate. The reason is the consciousness of the generality of the user preference problem and the desire to make the format popular among different distributions. As plain text is the universal encoding for information interchange formats in FOSS communities [12], using a plain text format makes it easy for package manager developers to adapt tools to CUDF.

3.1 CUDF Syntax

The CUDF encoding of an upgrade scenario assumes the name of *CUDF document*. Every such document has an abstract logical structure, a formal meaning, and a serialized form as a plain text file. The *logical structure* of a CUDF document—sketched in Figure 2—is based on *stanzas*, which are collections of key-value pairs called *properties*. Values are typed within a simple *type system* containing basic data types (e.g.: integers, boolean, and strings) and more complex, package-specific, data types such as boolean formulae over versioned packages used to represent inter-package dependencies.

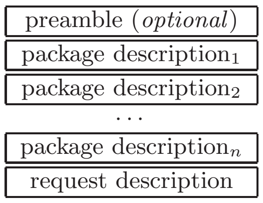

Figure 2: Overall structure of a CUDF document.

CUDF documents contain one *package description stanza* for each package known to the package manager; collectively they represent the *package universe*. This means that both installed and non-installed (but available) packages are represented in the same way in the same document, in contrast to current package installation systems which often distribute this information over different files using different syntactic representations.

Package description stanzas are based on a core set of properties (sometime optional, but always with default values), the most important of which are: `package` and `version` (which unambiguously identify packages), `depends` and `conflicts` (which express package dependencies and conflicts to be properly installed), `provides` (which expresses versioned *features* that the current package provides for other packages to depend or conflict upon), and `installed` (which state whether the current package is installed or not).

Figure 3 shows the serialization of a sample CUDF document. As stanzas are separated by blank lines, the central part of the figure shows three package description stanzas, starting with the `package` property, where both core and extra properties are used. The latter must be declared in the optional *preamble stanza*, which starts the document in Figure 3. The ability to declare extra properties accounts for extensibility and also enables to statically verify the syntactic correctness of CUDF documents. The bottom part of Figure 3 shows the *request description stanza*, where the user request is expressed. In its minimal form, such stanza is used to express which packages the user wants to `install`,

```
preamble:
property: suite: enum(stable,unstable) = \
 "stable"
property: bugs: int = 0

package: car
version: 1
depends: engine, wheel, door, battery
installed: true
bugs: 183

package: bicycle
version: 7
suite: unstable

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

...

request:
install: bicycle, gasoline-engine = 1
upgrade: door, wheel > 2
```

Figure 3: Sample CUDF document.

`remove`, or `upgrade` (using the homonym properties), possibly specifying version requirements.

The example lacks the encoding of user preferences. This lack, which was our initial motivation for the work reported here, can be filled by an optional property specifiable in the request stanza, called `preferences`. Its content is a MooML program, discussed in the next section. What is relevant here is that MooML programs may be part of CUDF documents and will be able to express preferences referencing CUDF stanzas.

3.2 CUDF Semantics

Given that a CUDF document completely describes an upgrade scenario, what does constitute its *meaning*? Intuitively, an upgrade scenario poses a challenge for the package manager, its solutions are new package statuses. The meaning, or semantics, of a CUDF document is hence a characterization of all *valid solutions* matching the upgrade scenario. We recall that a package status is just a set of packages contained in the package universe which we know is fully encoded in the document. On that basis, we declare that a solution is valid if and only if:

1. all installed packages have their dependencies satisfied, i.e. installed as well (*abundance*);

2. no two packages that are in conflict are installed together (*peace*);

3. the user request is satisfied by installed packages (*correctness*).

The first two points have been previously formalized relying on an encoding in propositional logics [9]. That encoding fails to respect the design principle of staying close to the original problem since, for example, packages with the same name and different versions are treated as unrelated boolean variables in the encoding. The formal semantics of CUDF characterizes all valid solution corresponding to a given CUDF document as a binary relation among package statuses, indexed by the user request. We will not give the full details here, for which the reader is referred to [14], but rather only discuss the *peculiarities* of CUDF formal semantics and its differences with respect to previous encodings.

An important semantic difference between existing package management systems in FOSS distributions is whether they a priori allow packages to be installed in multiple versions (like `rpm` does) or not (like `dpkg`). CUDF semantics here follows the `rpm` philosophy of a priori allowing multiple versions of a package to be installed at the same time. To encode Debian-like upgrade scenarios, where different versions of the same package are forcibly in conflict, a special case of conflicts semantics is exploited, namely: self-conflicts are ignored. Hence, in Figure 3, all packages (potentially) appearing in multiple versions declare an (unversioned) conflict with themselves, as it happens for `gasoline-engine`. The semantics ensures that such conflicts are ignored for the very same version of the package (otherwise those packages will be useless) but take effect on different versions of `gasoline-engine`, granting that only one version of it can be installed. Such a semantics is coherent with self-conflicts on virtual packages, which can be exploited to ensure mutual exclusions among different providers of the same feature. For instance, three packages like `postfix`, `sendmail`, and `qmail`, all providing the `mail-transport-agent` feature, can be made mutually exclusive by having all of them both provide *and* conflict with `mail-transport-agent`.

Finally, feature provision via `provides` is versioned, meaning that specific versions of a given feature can be provided. Not specifying a version—as in `provides: foo`—is interpreted as providing *all versions* of the `foo` feature.

Equipped with all this, verifying the satisfaction of a user request boils down to re-use the notions of peace and abundance: an `install` request is satisfied if and only if the same line, considered as a dependency, would be satisfied (abundance); a `remove` request is satisfied if and only if a corresponding conflict is unsatisfied (peace). Only `upgrade` needs some caution: in principle it can be handled as `install`, but additionally it also requires that all packages mentioned in the user request are installed in a single version. Furthermore, after `upgrade`-ing some package we must have a version of that package that is at least as new as any previously installed version of that package.

CUDF also allows to express that a particular package must not be removed, that it must be kept in its current version, or that its functionality must be provided by some package (see [14] for details).

3.3 CUDF Implementations

CUDF has already seen various implementations. The first implementation—`libcudf`—is the "reference" implementation of the CUDF specifications and has been developed by one of this paper authors. `libcudf` consists in a library able not only to parse and pretty print CUDF documents, but also to verify the CUDF semantics. This latter feature can be exploited in two ways:

1. given a CUDF document, `libcudf` can verify whether the contained package status is *consistent*, i.e., whether abundance and peace are verified for all its packages;
2. given a CUDF document and an encoding of a potential solution, `libcudf` can verify whether the solution is valid, i.e., abundance + peace + user-request satisfaction.

`libcudf` comes with the `cudf-check` command line tool which provides the above two features out of the box. The library is Free Software and can be user both from the OCaml and C programming languages; it is available for download at `http://www.mancoosi.org/software/`.

The authors are aware of other CUDF implementations. Some of them are being developed within MANCOOSI to convert distribution-specific upgrade scenario descriptions into CUDF, so that a cross-distribution corpus of upgrade scenarios can be formed. They will be released shortly at least for the following distributions: Mandriva, CaixaMagica, Debian GNU/Linux. Using such tools we have verified that the average size of an upgrade scenario encoded in CUDF is linear with the size of the origin package manager information and usually smaller.[2]

Another independent implementation is already available in CUPT[3], a new APT-compatible package manager for Debian. In CUPT, CUDF is used as a syntactic format to pipe upgrade scenarios to external solvers, so that upgrade planning can be decoupled from other package manager activities. Also, such a choice enables sharing more easily dependency solvers not only inter- and intra-distributions, but also with the scientific community.

4. EXPRESSING USER PREFERENCES

Having a rigorous description of upgrade scenarios, we can now devise our language to express user preferences. Our proposal for such a language—MOOML for MancOosi Optimization Meta[4]-Language—is described in this section. The design of the language needs to face two requirements that appear to be in mutual conflict:

Simplicity programs written in MOOML have to be interpreted by solver tools that will try to satisfy user preferences. Hence they should be as simple as possible in order to minimize the burden put on the developers of these tools.

Expressivity the MOOML language should allow to express sophisticated optimization criteria expressive enough to encode the scenario we have discussed.

The right choice of a language was to be found between two extremes. On one extreme a *Turing-complete programming language* with rich user-defined data structures and function definitions through unrestricted recursion. This extreme would provide maximum expressivity by definition, but would require tool developers to integrate an interpreter for a full-fledged programming language. On the other extreme a *restricted language* allowing only for simple combinations of optimization criteria for which a limited choice of common simple criteria is provided. This extreme would

[2]e.g. on a large Debian installation, using both testing and unstable package repositories for about 45'000 packages, the package manager information on disk amounts to 14 Mb and the corresponding CUDF document has 9 Mb.

[3]`http://wiki.debian.org/Cupt`

[4]The *meta* is inherited from the ML family of languages, for our purpose there is no distinguished meta level.

P	::=		***program***
		`( let x = e )*`	*definition*
		`( constraint e )?`	*constraint*
		`( (minimize | maximize) e )*`	*criteria*

Figure 4: Syntax of MooML programs

probably make life easy for tool implementers, but would be too limited in expressivity. It would also bear the risk of being obliged to continuously extend the choice of optimization criteria.

In order to find the right balance between these two extremes we made the following design choices for MooML:

- MooML allows to separately specify hard constraints that must be satisfied by "user-approved" solution, and optimization criteria.
- MooML does not allow to program directly an algorithm that compares alternative solutions[5]. Instead, the language allows to define how to compute a *measure* of solution quality. Two possible solutions are compared by comparing their respective measures. A MooML program specifies the polarity of each measure (i.e., whether it should be minimised or maximised). In case several measures are defined the program defines a strict priority hierarchy (technically this is a *lexicographic combination* of orders).
- MooML is a strongly typed functional language allowing for polymorphic types and inference of principal types.
- MooML does not allow for arbitrary use of recursion, and is deliberately not Turing complete. Instead it provides for a generic `fold`-like iterator over lists, which allows to program primitive recursive functions over lists.
- MooML does not allow to define custom data types.
- MooML does not have a mechanism to catch exceptions but allows to express execution errors.

4.1 MooML Programs

The high-level syntax and structure of a MooML program is sketched in Figure 4. Such a program is composed by a series of preparatory global definitions, meant to be reused in the remainder of the program. Then, two main parts compose a MooML program. The first is a *constraint*, that is a boolean expression which, when evaluated to `true`, indicates a solution considered acceptable by the user. Using a constraint users can exclude solutions that, in spite of being valid with respect to CUDF semantics, are undesirable for them. The second part is a list of *optimization criteria*, i.e. expressions of the language returning integers and tagged with a request to either minimize or maximize them over all otherwise valid solutions.

The syntax of MooML *expressions*, as given in Figure 5, has features borrowed from common functional programming languages. Expressions sport rich types such as records,

[5]as it happens, for example, with the `sort` function provided by the standard library of several programming languages

e	::=		***expressions***
		`x`	*variable*
	|	$\mathbb{C}_v$	*literal*
	|	`fun x -> e`	*abstraction*
	|	`e e`	*application*
	|	`()`	*unit*
	|	$(\mathtt{e}_1, \ldots, \mathtt{e}_n)$	*tuple*
	|	$\{\mathtt{l}_1 = \mathtt{e}_1, \ldots, \mathtt{l}_n = \mathtt{e}_n\}$	*record*
	|	`[ ]`	*empty list*
	|	`e :: e`	*list*
	|	`e.l`	*projection*
	|	$\mathtt{let}\ \mathtt{p} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$	*let binding*
	|	`match e with` $\mathtt{p}_i \Rightarrow \mathtt{e}_i \mid \cdots \mid \mathtt{p}_n \Rightarrow \mathtt{e}_n$	*pat. match*
p	::=		***patterns***
		`x`	*variable*
	|	$\mathbb{C}_v$	*constant*
	|	`()`	*unit*
	|	$(\mathtt{p}_1, \ldots, \mathtt{p}_n)$	*tuple*
	|	$\{\mathtt{l}_1 = \mathtt{p}_1, \ldots, \mathtt{l}_n = \mathtt{p}_n\}$	*record*
	|	`'x`	*enumeration*
	|	`[ ]`	*empty list*
	|	$\mathtt{p}_1 :: \mathtt{p}_2$	*list*
	|	`*`	*wildcard*
$\mathbb{C}_v$	::=		***CUDF literals***
		`true	false`	*booleans*
	|	`...	-1	0	1	...`	*integers*
	|	`"s"`	*strings*
	|	`'l`	*enumerations*
	|	`...`	*formulae, ...*

Figure 5: Syntax of MooML expressions

tuples and lists defined on top of the basic CUDF types, as well as expressive constructs such as pattern matching and (non-recursive) local definitions. The evaluation of a MooML program is a straightforward ML-style evaluation [4] with pattern matching [2]; overall it boils down to evaluate the constraint and optimization criteria expressions in an evaluation environment enriched with global definitions. Additionally, the environment is also enriched with:

- the MooML *standard library*, which provides the usual kit of functional programming functions and in particular the `fold` iterator (and some of its derivatives, like `map`, and `filter`) without which iterating over list data structures would be impossible within the language;
- the *package universe* `u` denotes a list of records representing all the package description stanzas of the CUDF document from which the MooML program originated. Each record contains one field for each package property, and can therefore be properly typed having around the CUDF preamble. However, the `installed` property gets split into two new properties:

`was-installed` (the same as the original `installed`, renamed for clarity) denotes whether the owning package *was installed* in the upgrade scenario presented to the package manager

`is-installed` denotes whether the owning package *is* *installed* in the proposed solution in the context of which the MooML program is being evaluated

- the *user request* `r` denotes a record corresponding to the user request stanza of CUDF.

The only way to express iteration over lists is to use the predefined `fold` function. An expression

```
(fold f [an ; ... ; a1] a0)
```

is evaluated as

```
(f an (f a(n-1) ... (f a1 a0) ... ))
```

An alternative way to describe its semantics is the following iterative pseudo-code:

```
r := a0;
foreach i in 1 .. n do r := (f ai r);
return r;
```

For instance, the standard library contains a definition of the `sum` function to sum up a list of integers:

```
let sum l = fold add l 0
```

and other functions like `filter`, `map`, `max`, etc. acting on lists can easily be defined the same way.

Note that all properties, except the `is-installed` property, of packages are given by the CUDF document on which a MooML program is applied. The input CUDF document describes the `was-installed` property of any package, it is the role of the MooML program to impose constraints on the possible `is-installed` properties of packages, and to calculate a score on any possible choice of `is-installed` properties of the packages.

Once the constraint and criteria expressions are fully reduced, there are enough information to know whether or not the solution should be discarded (constraint evaluated to `false`). If it is not the case, the different criteria values together denote a tuple that can be *lexicographically* compared with tuples coming from other candidate solutions to determine which of the two is to be preferred. Of course the lexicographic order should take into account the "polarity" of the criterion, i.e., whether it was a `minimize` or `maximize` request.

Types are not explicitly given in the syntax of the language, because they can be reconstructed in the style of Damas-Milner [5], obtaining principal types. The only source of ambiguity in the type system are record labels which, due to the CUDF ability to declare extra properties, may be not sufficient to unambiguously determine record types. While there seems to be no obstacles in extending the type system to account for them in the style of Remy [13], we have preferred to provide optional type ascriptions in the concrete syntax (not shown in Figure 5) to disambiguate the rare ambiguous cases.

4.2 Examples

MooML is expressive enough to account for all usage scenarios presented in Section 2, as we will show in the following. The need of program simplicity for solver implementers will be addressed by the partial evaluation mechanism in the next section.

Example 1 (minimize total installation size). *The "Hello, World!" equivalent in* MooML *is likely to be the widespread policy of minimizing the total installation size, very useful for embedded or otherwise constrained systems. It can be expressed as:*

```
let size pl =
  sum (map (fun p -> p.installed-size)) pl
minimize size
  (filter (fun p -> p.is-installed) u)
```

where `sum` is a library function summing up integers. The program simply states that the score to be minimized is the sum of the `installed-size` value (an extra property with the obvious meaning) of all packages installed in the proposed solution.

Example 2 (maximize package "freshness"). *The scenario requiring to maximize the number of packages installed at their most recent version can be expressed as follows.*

```
let is-recent p =
  forall
   (fun q -> (q.name != p.name)
          || (q.version <= p.version)
   u
maximize cardinality
  (fun p -> p.is-installed && is-recent p) u
```

`is-recent` is used as an auxiliary function to check whether a given package—given as its record—is the most recent version of all equally named packages; its implementation relies on `forall` which check the `true`-ness of a boolean predicate over a list of items (in this case, the package universe `u`). Complementary, `cardinality` counts *the number of times a predicate is `true` over a list; in the given optimization criteria, it is used to require the maximization of "recent" packages.*

Example 3 (flexible APT pinning). *APT pinning can be encoded in at least a couple of different ways using* MooML, *depending on the desired goal. A first possibility is to encode the* exact *semantics of pinning, so that the only acceptable solutions will be those potentially returned by a pinning implementation. In essence, pinning works at the package choice level, ensuring that among all available versions of a given package, the one with the highest* pin priority *is installed. While pin priority themselves can be assigned using* MooML *(see Section 5), if we assume that each package comes with an extra property `pin-priority`, we can encode pinning semantics as follows:*

```
let max-pin p =
  max (map (fun z -> z.pin-priority)
         (filter (fun q -> q.name == p.name) u))
constraint forall (fun p -> p.is-installed
             && p.pin-priority = max-pin p)
```

Given that this strict semantics is a well-known cause of APT incompleteness [8], a more "flexible" pinning encoded can be obtained by requiring to maximize the number of packages at maximal pin priority:

```
let max-pin p = (* as above *)
maximize cardinality
  (fun p -> p.is-installed
          && p.pin-priority = max-pin p)
```

An even more flexible metric over APT pinning can be obtained by minimizing the total difference between maximal and actual pin priorities as follows:

```
let max-pin p = (* as above *)
minimize sum
  (map (fun p -> if p.is-installed
            then max-pin p - p.pin-priority
            else 0) u)
```

Example 4 (priority to security updates). *The scenario which requires to prioritize security upgrades over any other criteria can be encoded straightforwardly by relying on MooML's lexicographic ordering over solution measures. In the following example it is combined with the freshness criteria of Example 2.*

```
let is-recent p = (* as above *)
maximize cardinality
  (fun p -> p.is-installed && not p.was-installed
            && p.is-security-fix) u
maximize cardinality
  (fun p -> p.is-installed && is-recent p) u
```

Note that we explicitly require the package to be newly installed before verifying whether it is a security fix (extra property `is-security-fix`), this way we ensure the security fix is being delivered with the proposed solution. Lexicographic ordering ensures that solutions with a higher number of security fixes being delivered will be preferred, no matter the total freshness. (How to improve the example to ensure that no past security fixes get removed by downgrades is left as an exercise.)

Example 5 (minimize multiple versions). *In this example we wish to minimize the number of packages that exist in multiple versions in the final installation.*

```
let number-versions p = length
   (filter (fun q -> q.is-installed &&
                     p.name = q.name)
           u)
minimize cardinality
   (fun p -> p.is-installed &&
             number-versions p > 1) u
```

The function `number-versions` applied to package p calculates the number of installed packages with the same name as the package p. We minimize the number of installed packages for which the function `installed-version` returns a value strictly greater than 1.

5. PARTIAL EVALUATION

In their full generality, MooML programs can be too complex to handle for dependency solvers, or at least require non trivial implementation efforts to develop a full MooML evaluator. To address this shortcoming, MooML has been designed to be a good subject for *partial evaluation* which processes fully general MooML programs and returns "simpler" programs, ideally more suitable for digestion by dependency solvers. More precisely (see Figure 6), MooML partial evaluation is applied to a program p, which belongs to a CUDF document c, and returns two new entities: a *new program* p' and a *CUDF transformer* applicable to "c-like" CUDF documents, intuitively documents sharing the same extra properties of c. Once applied to c, the transformer returns a new document c', to which p' belongs. Partial evaluation enjoys the property that the evaluation of p in the context of c returns the same result (constraint and measure tuple) than the evaluation of p' in the context of c'. The advantage of p' over p is that it is potentially simpler, in the sense that

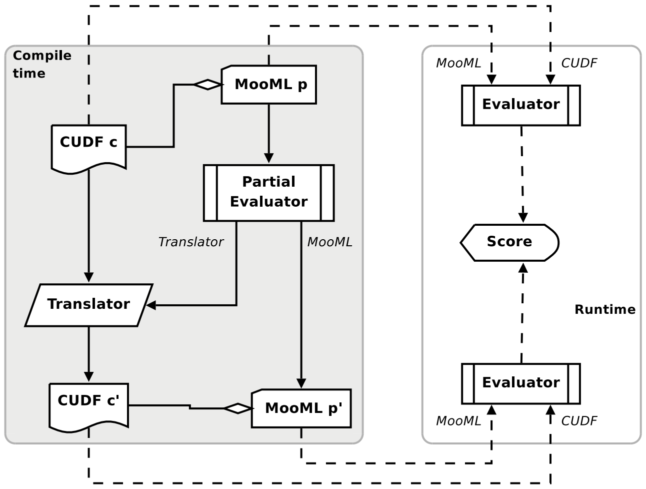

Figure 6: Partial evaluation and its properties

it can be implemented by ignoring significant parts of the MooML language. However, in the worst case, the partial evaluator may not be able to do any simplification.

The guiding principle of MooML's partial evaluation is to pre-compute all sub-expressions that depend on the upgrade scenario, but not on the upgrade solution, and to "save" them as fresh package properties. As a consequence, p' is obtained by substituting complex sub-expressions with access to (fresh) properties, and c' is obtained by adding (fresh) properties. To characterize a little more formally, the sub-expressions that are good partial evaluation candidates we first define an equality relation which relates all package statuses equal up to `is-installed`:

Definition 1 (sibling package lists). *Two package lists l_1, l_2 are* siblings, *written $l_1 \backsimeq l_2$, if $Dom(l_1) = Dom(l_2)$ (i.e., they contain the same packages), and for each $(p, v) \in Dom(l_1)$ we have that $l_1(p, v)$ equals $l_2(p, v)$ except possibly the value of the `is-installed` property.*

Then, we grasp partial-evaluable (sub-)expressions with the notion of local expressions. In the following definition we will make use of a mathematical semantics of the MooML language (the formal definition of which is omitted from this paper). When e is a MooML expression and σ an evaluation environment mapping identifiers to semantic values, then $[[e]]\sigma$ denotes the semantic object obtained by evaluating e in the environment σ.

Definition 2 (local expressions). *An expression e of type `package` $\rightarrow$ `t`, and which does not have any unbound identifiers besides `r` and `u`, is called* local *if for all packages p, for all package lists l_1, l_2, request r_0 such that $l_1 \backsimeq l_2$ and $p \in l_1$, $p \in l_2$ we have that*

$$[[e]][\mathtt{u} \mapsto l_1, \mathtt{r} \mapsto r_0](p) = [[e]][\mathtt{u} \mapsto l_2, \mathtt{r} \mapsto r_0](p)$$

Intuitively, local expressions are all those expressions whose evaluation does not depend on the `is-installed` values of packages coming from the package universe; note that expressions accessing the `is-installed` property of their argument can be local nevertheless. As stated in Definition 2, the expression e must not refer to any previously defined function, but this is not really a restriction as we can always inline all function definitions (since the language does not allow for recursive definitions).

We extend the MooML type system in order to *determine* a set of expressions that are local in the sense of Definition 2. The extension is straightforward and in the style of Volpano [18]. The record type gets split into *safe* and *unsafe* records, with type instantiation that enables to "cast-down" safe to unsafe; complementary, record projection typing gets changed to type as unsafe record projections explicitly accessing the `is-installed` property. The intuition is that functional expressions having *principal* type with safe record argument are guaranteed not to access its `is-installed` property.

Equipped with the above typing machinery, each MooML sub-expression—no matter where it appears—that have type `package` → `t`, for some `t`, can be tested for locality as follows:

1. if it can be typed under the premise that `u` is a list of *safe* packages, then the expression is local
   - (a) if, moreover, its principal type is an arrow from *safe* packages to something, the expression is fully determined without the candidate solution
   - (b) otherwise, the expression depends on the property `is-installed` of its sole argument
2. otherwise the expression is not local

Case (1a) is the luckiest: the sub-expression can be pre-computed on all packages of the universe, its value stored in a fresh property name (to be declared in the preamble), and replaced by a field access the fresh property. Case (1b) requires the additional efforts of (statically) computing *two* possible values of the sub-expression, according to the possible values of `is-installed`, and of tweaking the program to lookup one or another fresh property according to the actual `is-installed` value at runtime. Case (2) is the worst case, where no partial evaluation is possible due to non locality.

EXAMPLE 6. *To demonstrate partial evaluation in practice we reconsider Example 2. It contains two expressions having types* `package` → `t` *for some t. The first one, sub-expression of the is-recent definition body, belong to case (1a) (unrelated to solution), while the second one needs the property is-installed of its argument, still being local. Partial evaluation will rewrite the* MooML *program leading to something like:*

```
let is-recent p = forall (fun q -> q.fresh0) u
maximize cardinality
  (fun p -> if p.is-installed
            then q.fresh1
            else q.fresh2)
```

where fresh0, fresh1, fresh2 are fresh properties defined as follows. fresh0 will be true for all "most recent packages", fresh1 will inherit from fresh0, and fresh2 will be the constant false.

The limits of the partial evaluation approach are demonstrated by the example of minimizing multiple installed versions of packages (Example 5). In that case the partial evaluator does not bring any advantage since everything depends on the final installation status of the packages, and there is no additional information that can be pre-computed independently of the installation status of the *other* packages in the universe. A similar case is the maximization of the number of installed packages that have their `recommends` (which is a weak, non-mandatory form of package dependency) satisfied.

A concluding noteworthy scenario is a reprisal of APT pinning handling (see Example 3). No matter how "strictly" pinning gets implemented in MooML, partial evaluation enables to relax the requirement that pin priorities reach MooML pre-computed, without neither implementation burden, nor performance loss for dependency solvers. The idea is to store in MooML the rules to assign pin priorities to packages on the usual basis (origin suite, package name, package version, . . .) relying on apposite extra properties and suitable standard library functions (like regular expression matching). If pinning assignment is encoded as functions from packages to integers (and hardly will be otherwise), there is no reason for the implementing expression to access the `is-installed` property, given that pinning rules are static. Hence, the resulting sub-expressions are local—case (1a)—and will be completely removed during partial evaluation, returning a CUDF document such as those assumed by Example 3.

6. CONCLUSION

The request to alter the installation of component based software collections as large as FOSS distributions can have a daunting number of satisfying answers. To choose the "best" solution among them, state of the art package managers implement ad-hoc heuristics and offer preference mechanisms of limited expressiveness. In this paper we presented an architecture to specify user preferences about FOSS packages which is both independent from specific package managers or distributions and expressive enough to encode several preference scenarios. The architecture is composed by a format to encode upgrade scenarios (CUDF) and by a functional language to encode user preferences (MooML).

Future work is planned on several directions. First of all, while syntax and formal semantics of CUDF have been studied already, various properties of MooML still need to be investigated in more detail. In particular we plan to characterize various subsets of MooML that correspond, after partial evaluation, to language fragments which are best suited for different encodings of package upgrade problems (SAT, PBO, constraint programming, etc.).

We also plan to carry partial evaluation further in the direction of getting rid of data types at partial evaluation stage, so that only integers (which are the preferred data type for the optimization community) remain after that.

Finally, the mentioned corpus of upgrade scenarios coming from different distributions is actually being collected with the final goal of organizing a recurrent dependency solving competition. Ideally, such a that forum will become a venue where the package manager developer community meets the research community on constraint solving. Both communities could profit from this: package managers can use complete and more powerful dependency solving tools, and it gives the research community access to a large corpus of real-life optimization problems of non-trivial size.

7. REFERENCES

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *Empirical Software Engineering and Measurement 2009*, 2009. To appear.

[2] L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 368–381. Springer-Verlag, 1985.

[3] D. L. Berre and P. Rapicault. Dependency management for the Eclipse ecosystem. In R. D. Cosmo and P. Inverardi, editors, *IWOCE 2009 (this volume)*, Aug. 2009.

[4] D. Clément, T. Despeyroux, G. Kahn, and J. Despeyroux. A simple applicative language: mini-ml. In *Conference on LISP and functional programming*, pages 13–27, New York, 1986. ACM.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, USA, 1982. ACM.

[6] R. Di Cosmo and S. Cousin. Project presentation. Deliverable D1.1, The Mancoosi project, Jan. 2008. http://www.mancoosi.org/deliverables/d1.1.pdf.

[7] R. Di Cosmo, P. Trezentos, and S. Zacchiroli. Package upgrades in FOSS distributions: details and challenges. In *HotSWUp'08*, pages 1–5. ACM, 2008.

[8] EDOS Project. Report on formal management of software dependencies. EDOS Project Deliverable D2.1 and D2.2, Mar. 2006.

[9] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE 2006*, pages 199–208, Tokyo, Japan, Sept. 2006. IEEE CS Press.

[10] G. Niemeyer. Smart package manager. http://labix.org/smart, 2008.

[11] G. Noronha Silva. APT howto. http://www.debian.org/doc/manuals/apt-howto/, 2008.

[12] E. S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 1st edition, Oct. 2003.

[13] D. Rémy. Type inference for records in natural extension of ml. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical aspects of object-oriented programming*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.

[14] R. Treinen and S. Zacchiroli. Description of the CUDF format. Deliverable D5.1, The Mancoosi project, Nov. 2008. http://www.mancoosi.org/deliverables/d5.1.pdf.

[15] R. Treinen and S. Zacchiroli. Solving package dependencies. In DebConf8 Proceedings Team, editor, *DebConf 8*, pages 18–42, Mar del Plata, Argentina, Aug. 2008. https://media.debconf.org/dc8/proceedings/proceedings.pdf.

[16] P. Trezentos, R. Di Cosmo, S. Lauriere, M. Morgado, J. Abecasis, F. Mancinelli, and A. Oliveira. New Generation of Linux Meta-installers. *Research Track of FOSDEM 2007*, 2007.

[17] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *ICSE '07*, pages 178–188. IEEE Computer Society, 2007.

[18] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, 1996.